Partitioning
and Divide-and-Conquer Strategies

Partitioning

Partitioning simply divides the problem into parts.

Divide and Conquer

Characterized by dividing problem into sub-problems of same form as larger problem. Further divisions into still smaller sub-problems, usually done by recursion.

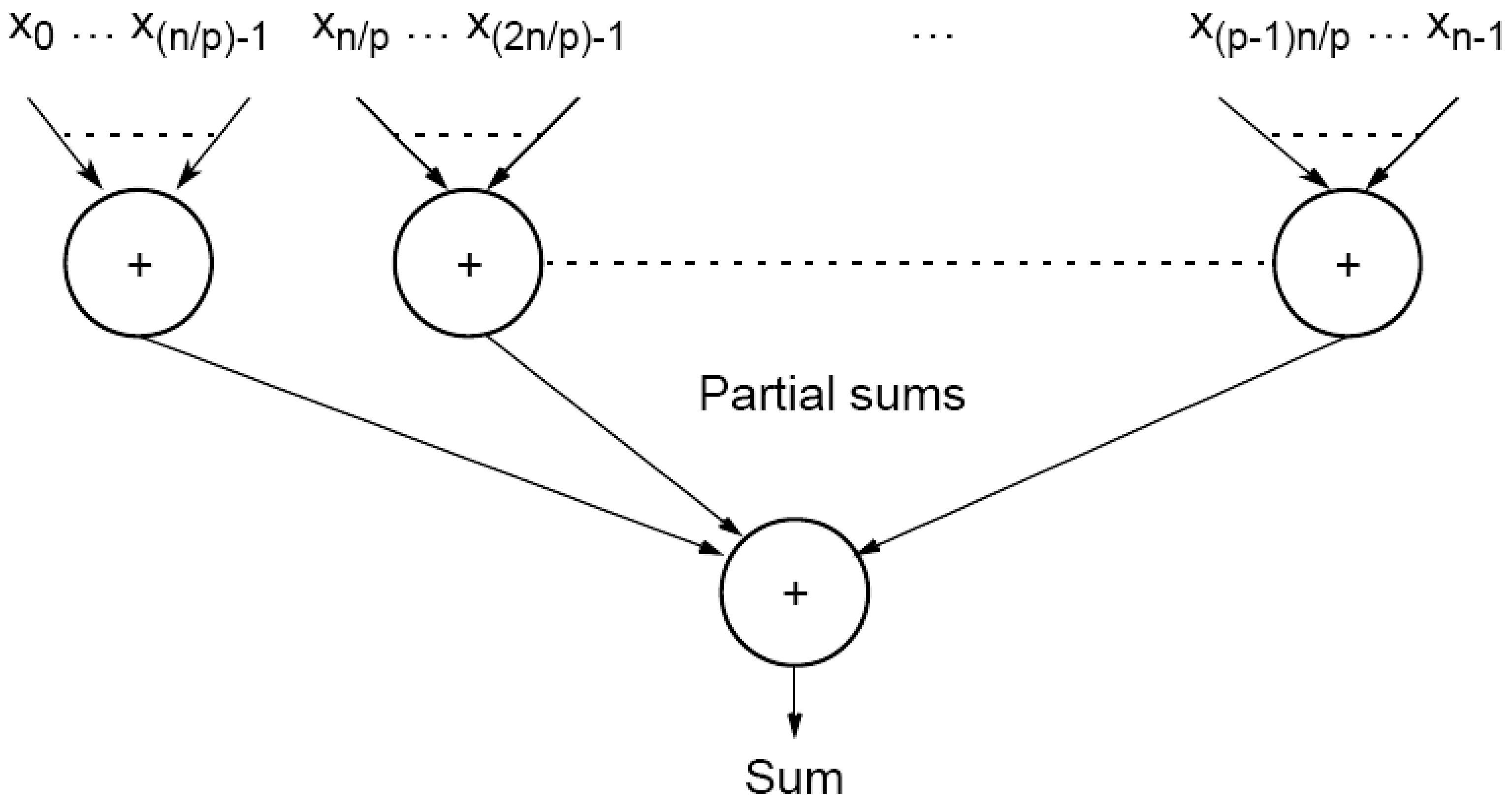
Recursive divide and conquer amenable to parallelization because separate processes can be used for divided parts. Also usually data is naturally localized.

Partitioning/Divide and Conquer Examples

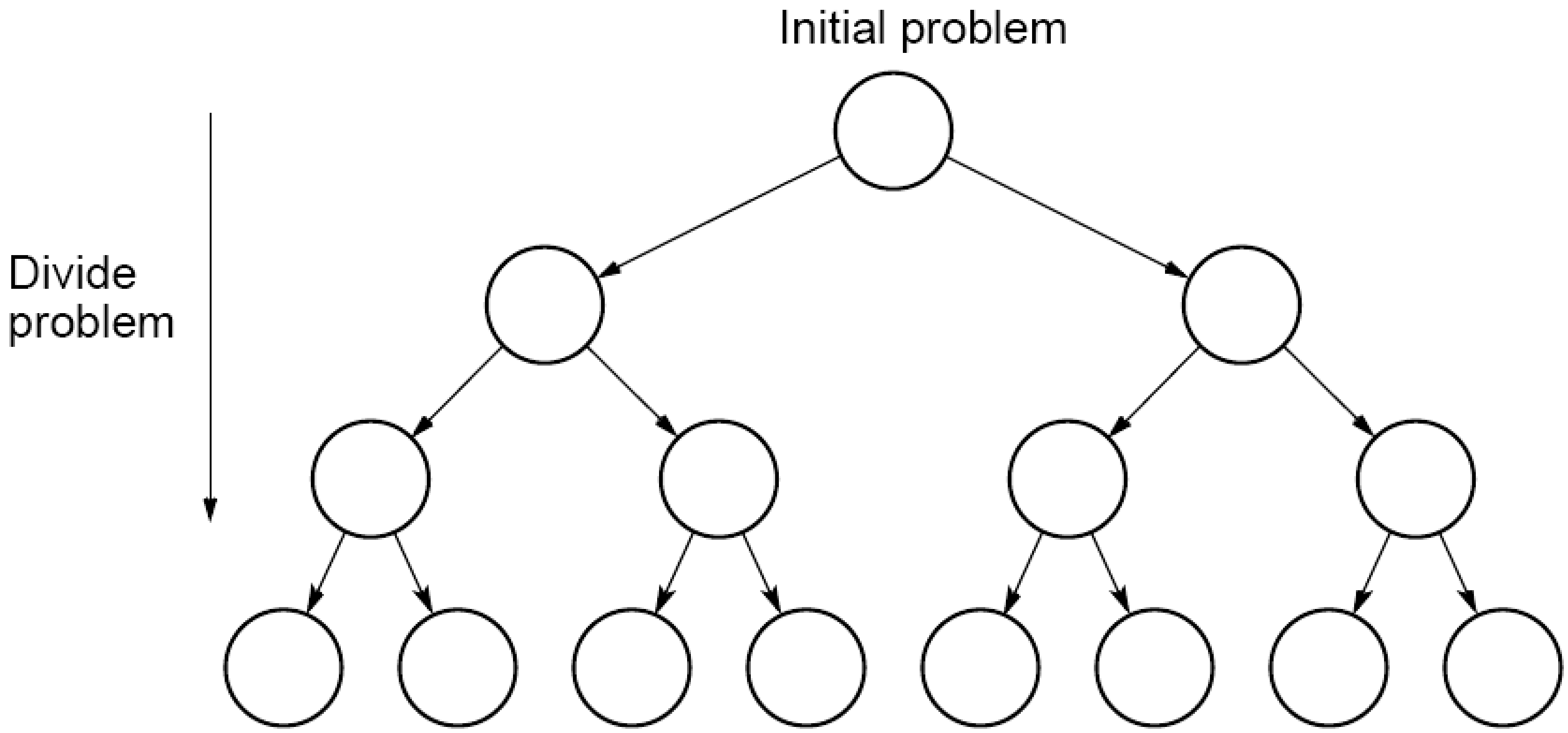
Many possibilities.

- Operations on sequences of number such as simply adding them together
- Several sorting algorithms can often be partitioned or constructed in a recursive fashion
- Numerical integration
- *N*-body problem

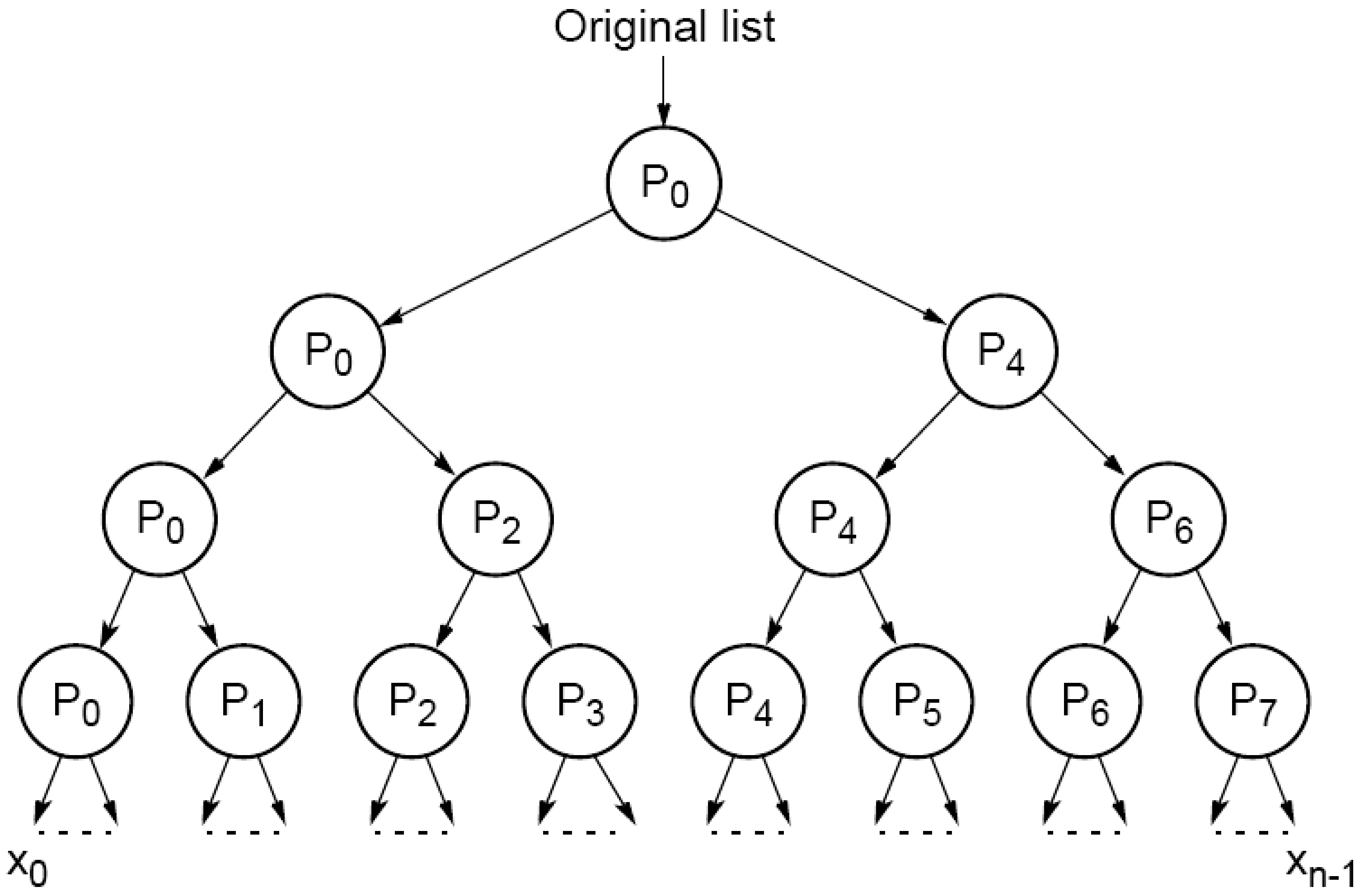
Partitioning a sequence of numbers into parts and adding the parts



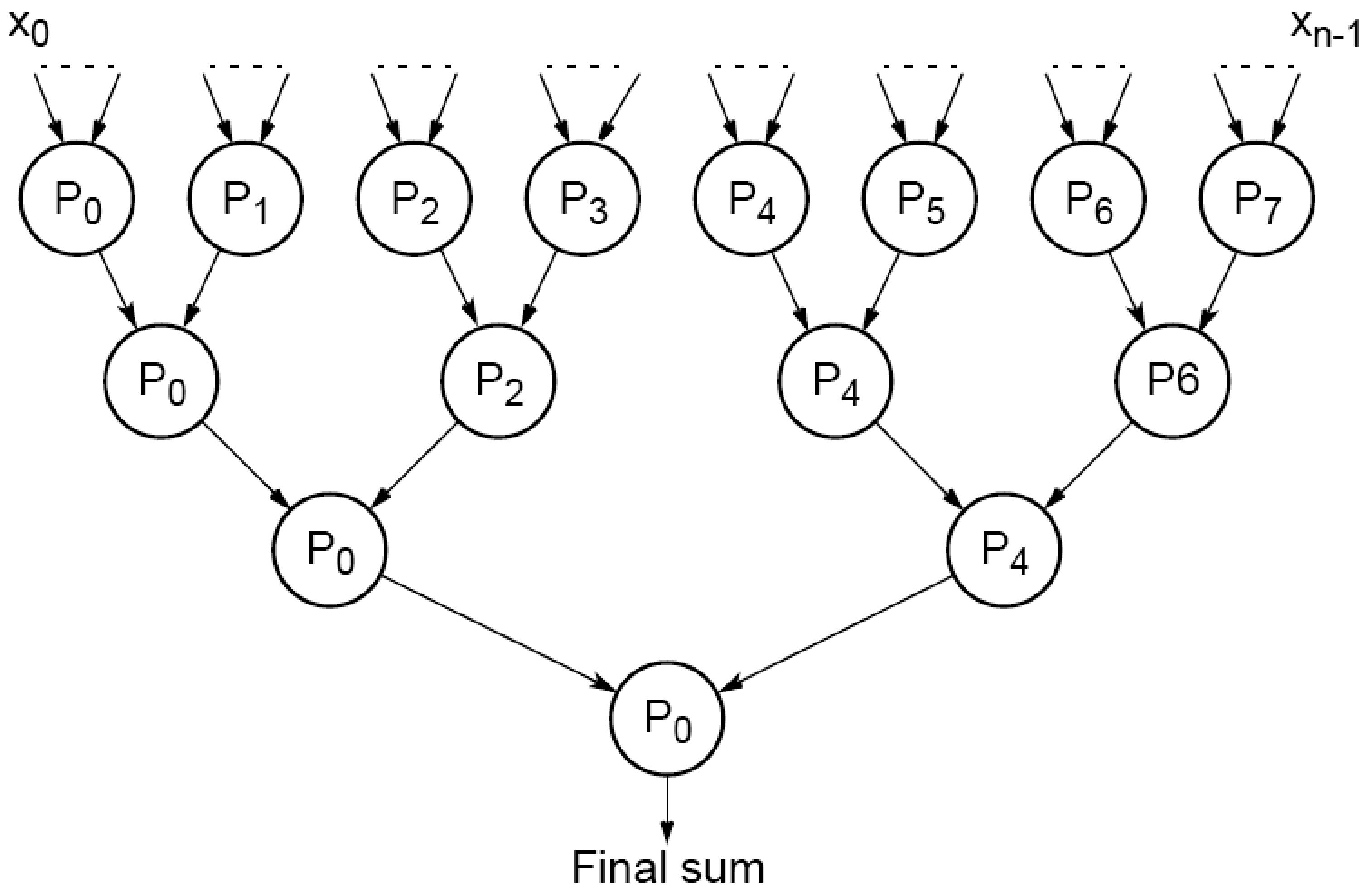
Tree construction



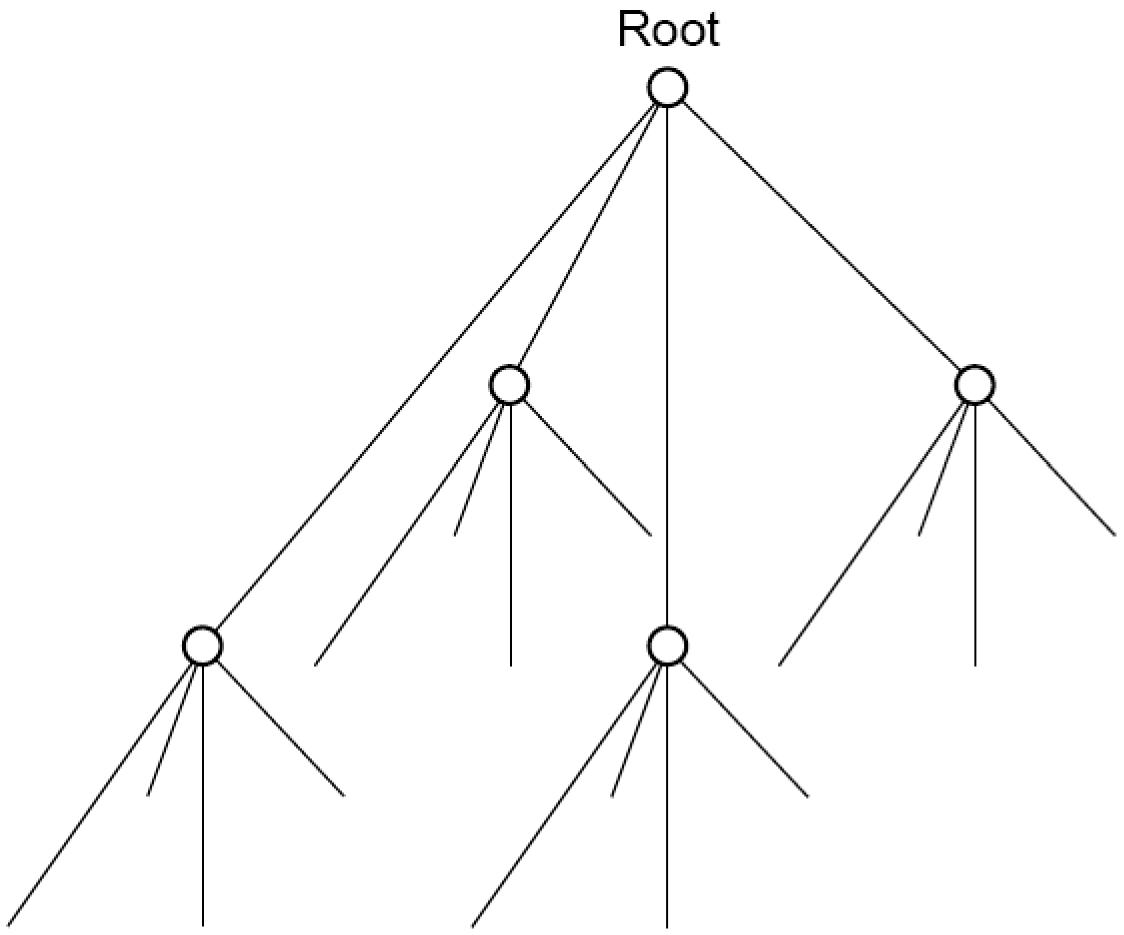
Dividing a list into parts



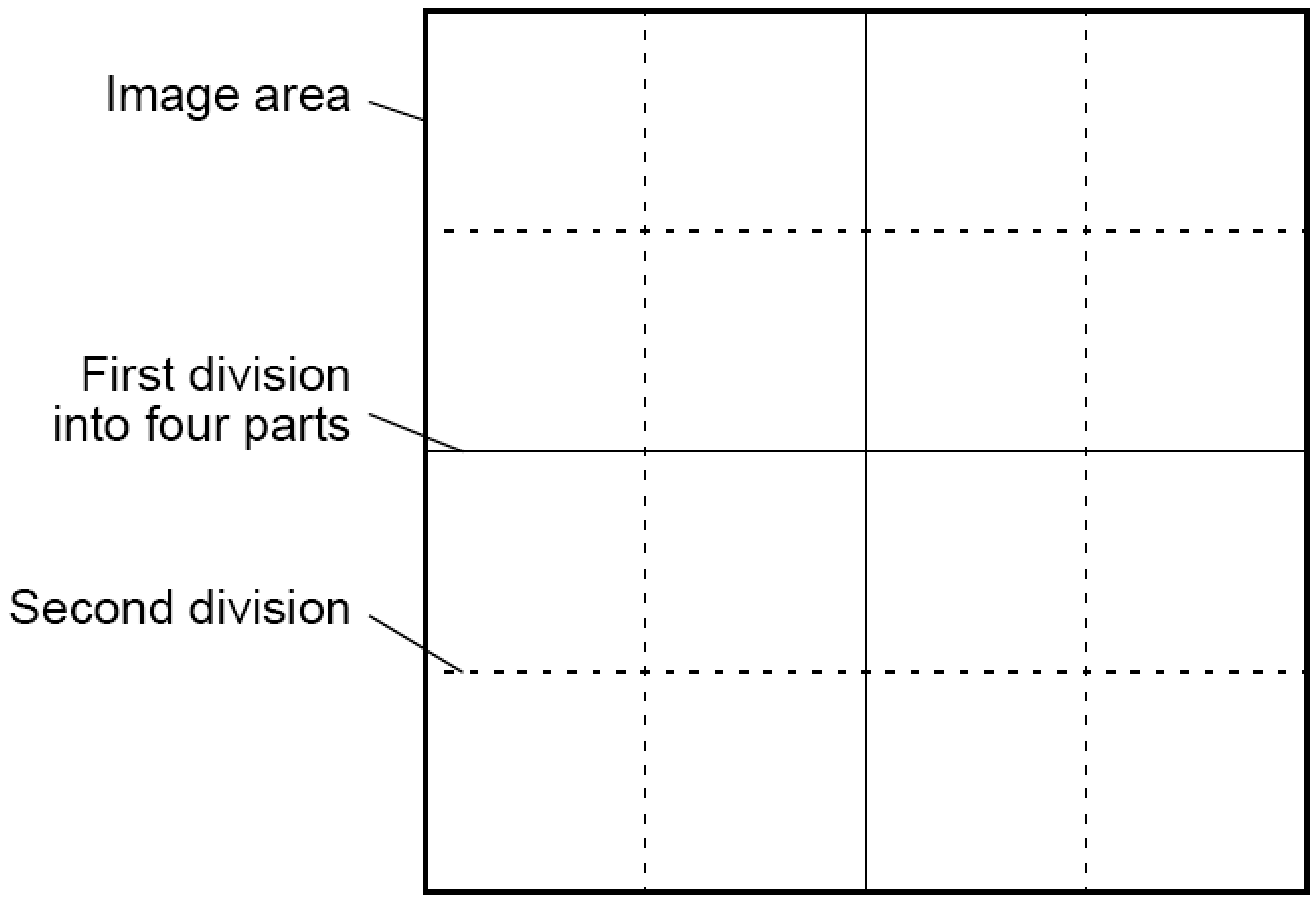
Partial summation



Quadtree

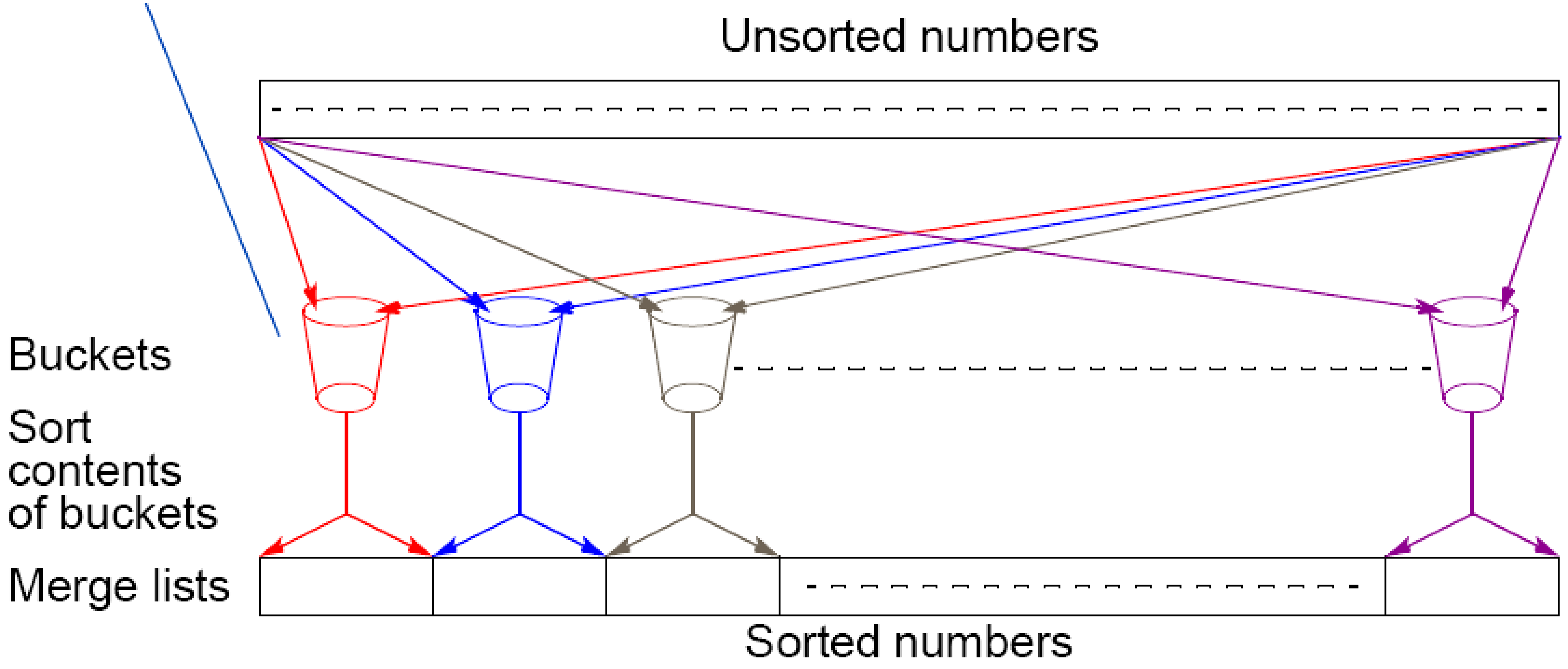


Dividing an image



Bucket sort

One "bucket" assigned to hold numbers that fall within each region.
Numbers in each bucket sorted using a sequential sorting algorithm.

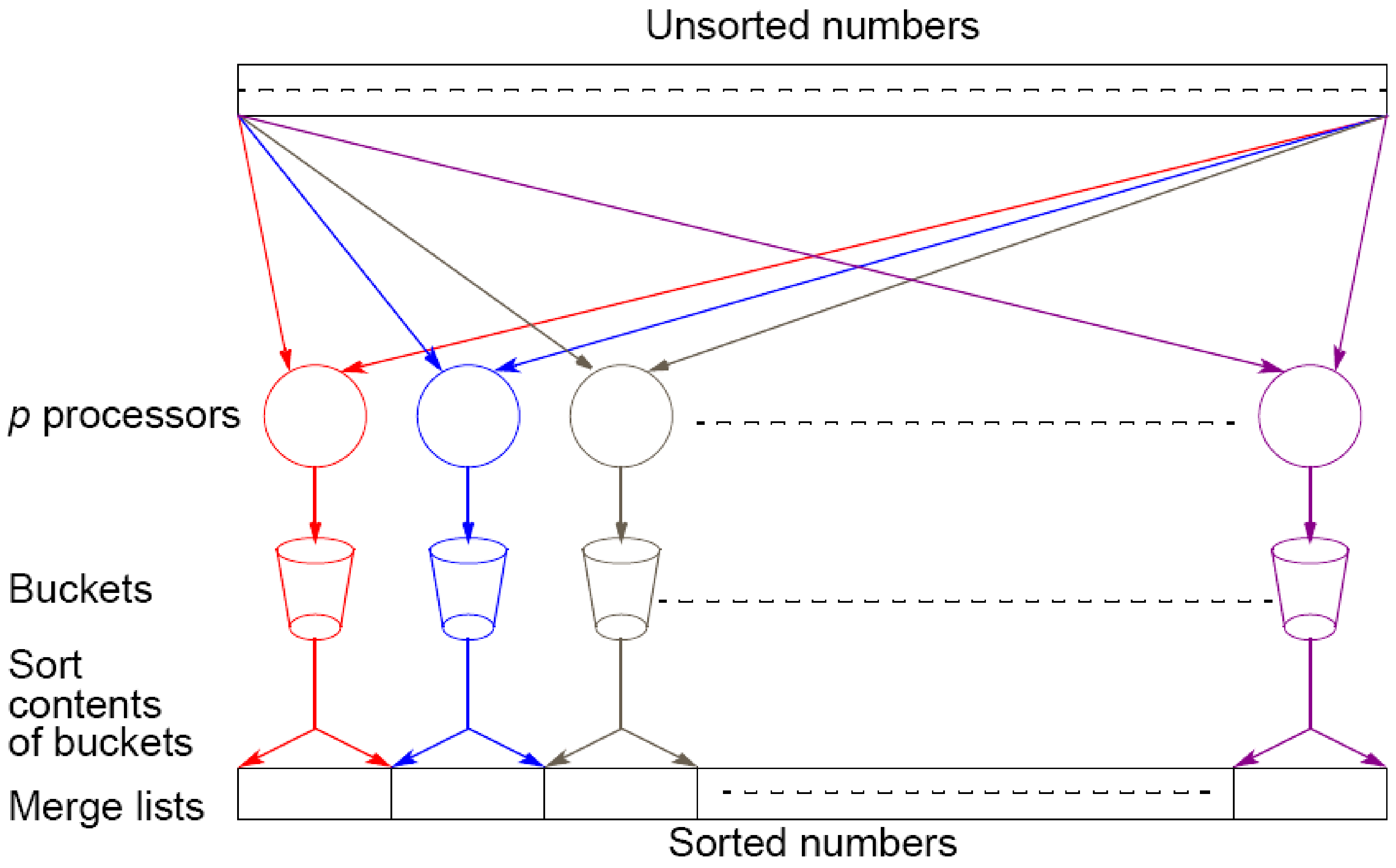


Sequential sorting time complexity: O(nlog(n/m)).
Works well if the original numbers uniformly distributed across a known interval, say 0 to *a* - 1.

Parallel version of bucket sort

Simple approach

Assign one processor for each bucket.



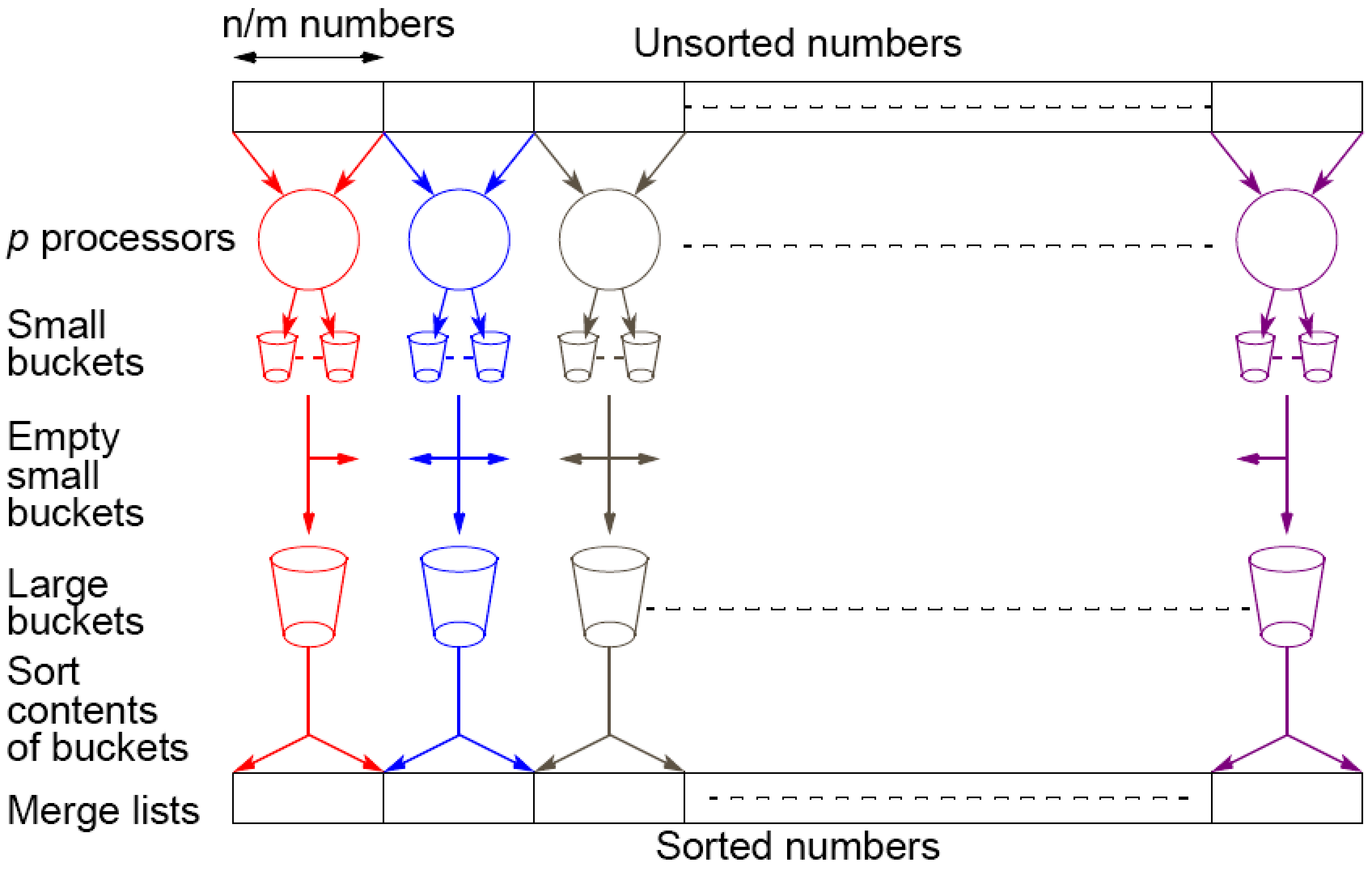
Further Parallelization

Partition sequence into *m* regions, one region for each processor.

Each processor maintains *p* “small” buckets and separates numbers in its region into its own small buckets.

Small buckets then emptied into *p* final buckets for sorting, which requires each processor to send one small bucket to each of the other processors (bucket *i* to processor *i*).

Another parallel version of bucket sort

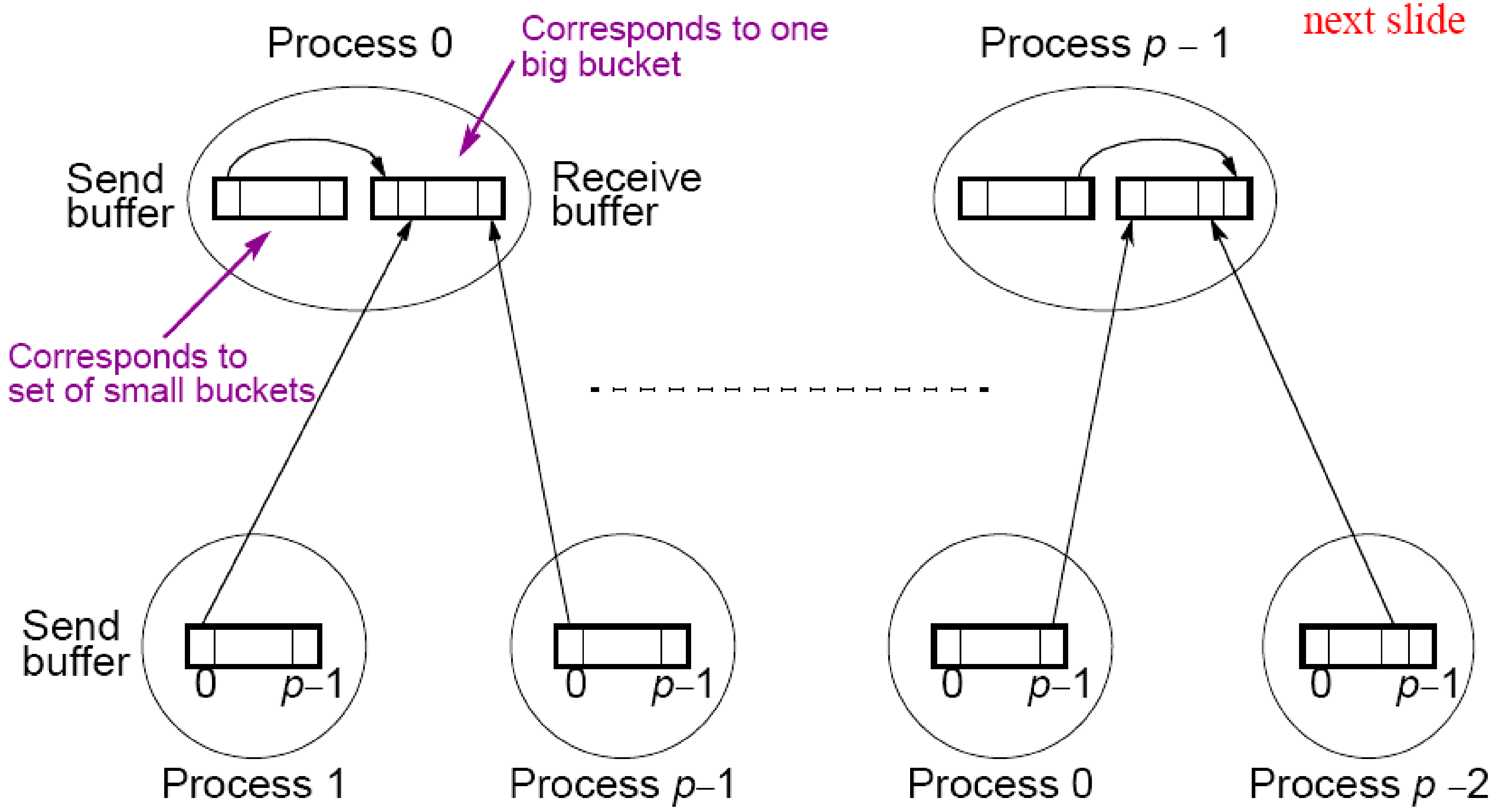


Introduces new message-passing operation - all-to-all broadcast.

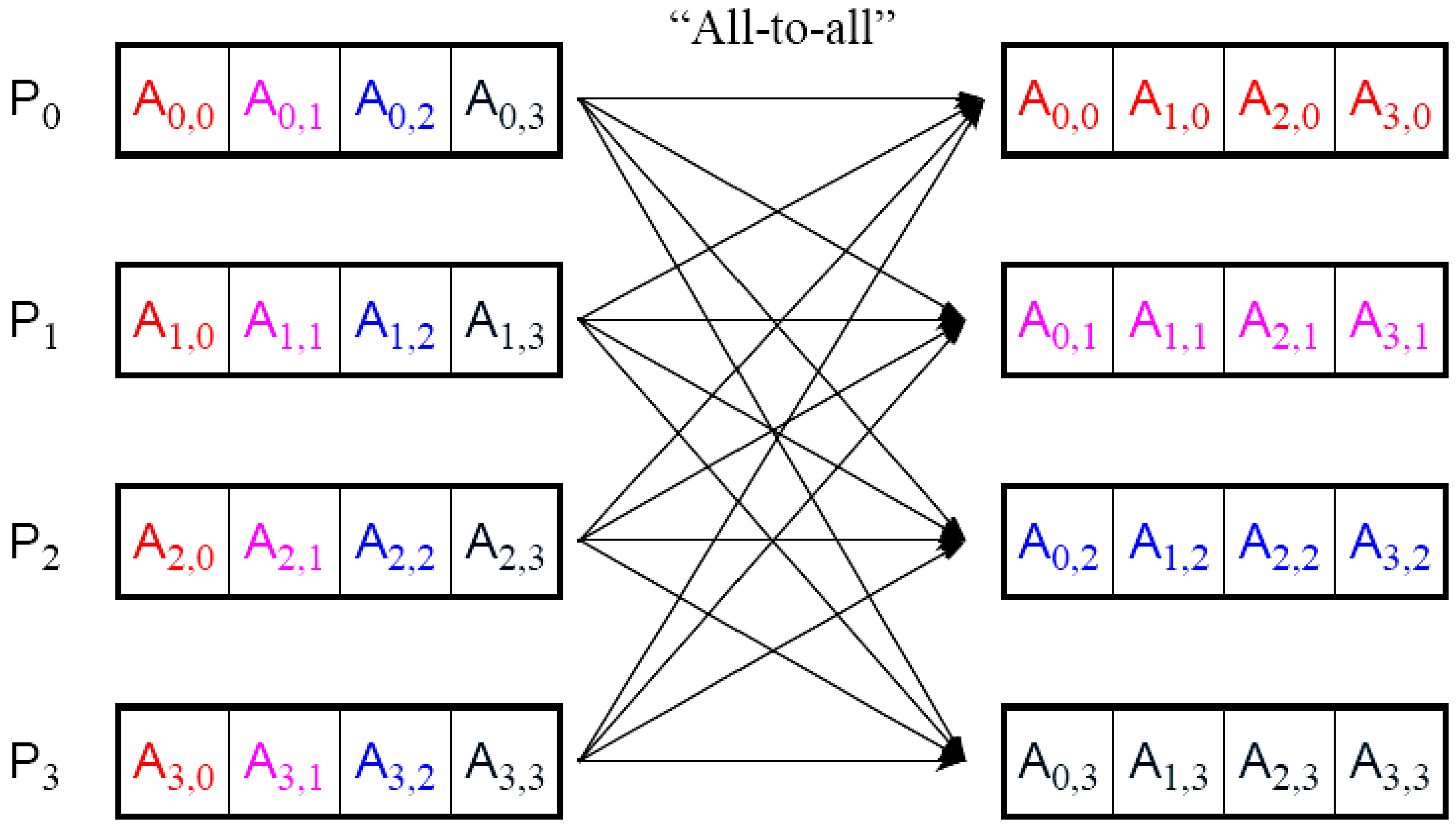
"all-to-all" broadcast routine

Sends data from each process to every other process

See also next slide



"all-to-all" routine actually transfers rows of an array to columns: Transposes a matrix.


"All-to-all"
P_0
P_1
P_2
P_3
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
$A_{0,0}$ $A_{1,0}$ $A_{2,0}$ $A_{3,0}$
$A_{0,1}$ $A_{1,1}$ $A_{2,1}$ $A_{3,1}$
$A_{0,2}$ $A_{1,2}$ $A_{2,2}$ $A_{3,2}$
$A_{0,3}$ $A_{1,3}$ $A_{2,3}$ $A_{3,3}$

Performance Analysis

Broadcast groups of numbers to each processor

$$t_{comm1} = t_{startup} + nt_{data}$$

Separate *n/p* numbers into *p* buckets $t_{comp2} = n / p$

Distribute each bucket with *n/(p*p)* elements. Each process sending *(p-1)* buckets- worst case

$$t_{comm3} = p(p-1)(t_{startup} + (n / p^2)t_{data})$$

Overlapping communications leads to

$$t_{comm3} = (p-1)(t_{startup} + (n / p^2)t_{data})$$

Note this p is m the bucket number in serial case

Computation $$t_{comp4} = (n/p)\log(n/p)$$

Summing all these gives

$$t_p = (n/p)(1 + \log(n/p)) + pt_{startup} + (n + p - 1)(n/p^2))t_{data}$$

Speedup factor =

$$S(n,p) = \frac{n + n\log(n/p)}{(n/p)(1 + \log(n/p)) + pt_{startup} + (n + p - 1)(n/p^2)t_{data}}$$

Efficiency = *S(n,p)/p*

$$\approx \left[1 + \frac{pt_{startup} + nt_{data}(1 + 1/p)}{(n/p)(1 + \log(n/p))} \right]^{(-1)}$$

Plot this!

Constant efficiency as problem and processor sizes change is known as Isoefficiency

Can we pick problem sizes, n, and processor numbers, p, so that **Efficiency** = *S(n,p)/p* is **Constant ?**

$$\left[1+\frac{pt_{startup}+nt_{data}(1+1/p)}{(n/p)(1+\log(n/p))}\right]^{(-1)} = \text{const}$$

$$if\ \ n=p^2\ \ then\ \left[1+\frac{t_{startup}+pt_{data}}{(1+\log(p))}\right]^{(-1)} \approx\ \text{const}\quad ?$$

$$if\ \ n=p^3\ \ then\ \left[1+\frac{(t_{startup}/p)+pt_{data}}{(1+2\log(p))}\right]^{(-1)} \approx\ \text{const}\,?$$

Efficiency = *S(n,p)/p* is **Constant ?**

$$\left[1+\frac{pt_{startup}+nt_{data}(1+1/p)}{(n/p)(1+\log(n/p))}\right]^{(-1)} = \quad \text{const}$$

$$\frac{1}{1+\alpha}$$ Constant or increasing requires α decreasing

$$(n/p)(1+\log(n/p)) \geq pt_{startup}+nt_{data}(1+1/p)$$

$$n(1+\log(n/\mathrm{p})) \geq \mathrm{p}^2 t_{startup}+nt_{data}(\mathrm{p}+1)$$

Let $n = p^{\beta}$

$$p^{\beta}(1+(\beta-1)\log(\mathrm{p})) \geq p^2 t_{startup}+p^{\beta}t_{data}(\mathrm{p}+1)$$

Hence $p^{\beta-2} > t_{startup}$ and $(\beta-1)\log(p) > t_{data}(\mathrm{p}+1)$

If startup dominates $\beta > 2$ but not really scalable

Scalability Implications

- Hence increasing the core count **p** by a factor of 1000 means that **n** the problem size has to grow by at least factor of as much as 1000,000 but only if startup cost dominates.
- The second β term is more problematic as we require roughly $\beta > p/\log(p)t_d$
- This is unlikely for large p.
- Is the algorithm scalable in terms of efficiency?

Bucket Sort Efficiency% t_{start} =1000 t_{data} =50

n/p	2	4	8	16	32	64	128	256	512	1024
10**4	**11**	**5**	**3**	**1**						
	12	**6**	**3**	**1**						
10**6	**14**	**7**	**4**	**2**						
	16	**8**	**4**	**2**						
10**8	**17**	**9**	**5**	**2**	**1**					
	19	**10**	**5**	**3**	**1**					
10**10	**20**	**11**	**6**	**3**	**1**					
	22	**12**	**6**	**3**	**2**					
10**12	**23**	**13**	**7**	**3**	**2**					
	25	**14**	**7**	**4**	**2**					
10**14	**26**	**15**	**8**	**4**	**2**					

Bucket Sort Efficiency $t_{startup} = 100$ $t_{data} = 5$

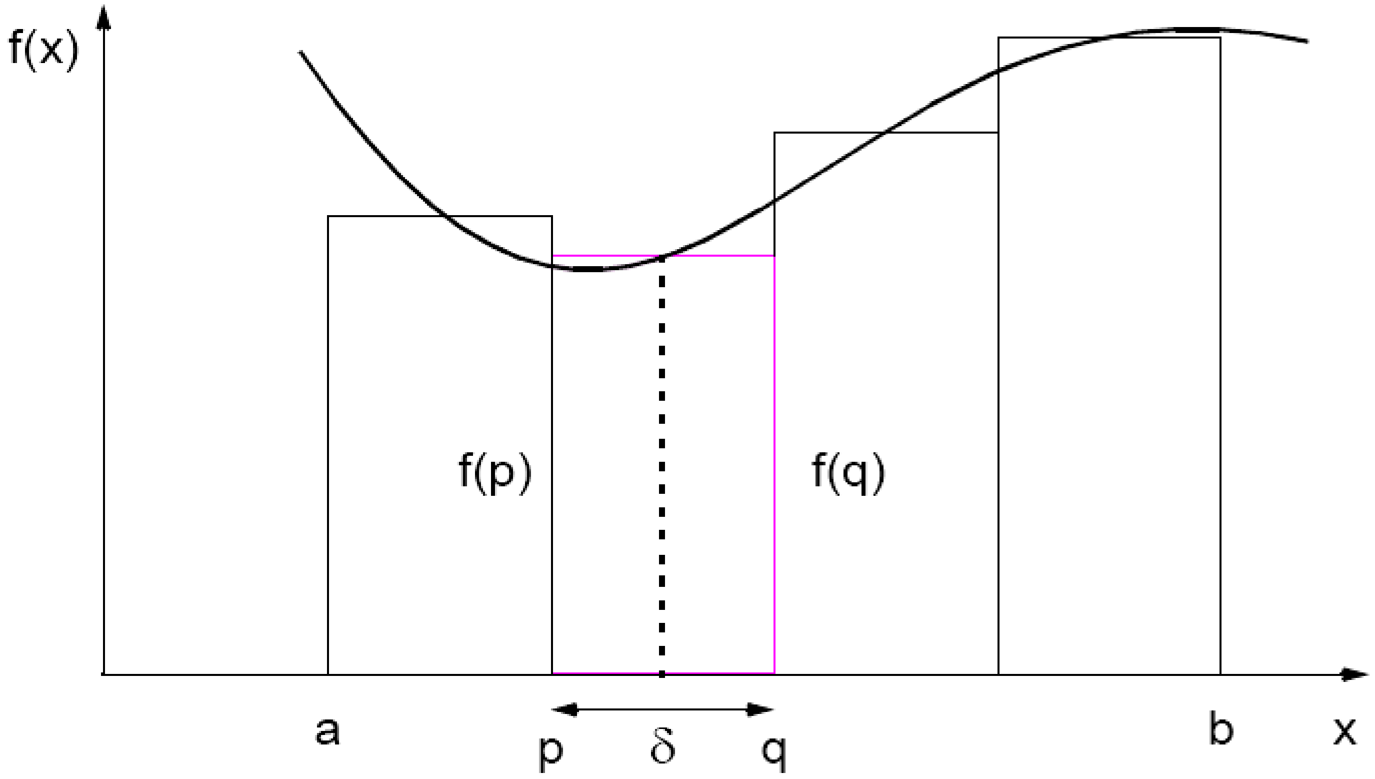
n/p	2	4	8	16	32	64	128	256	512	1024
10**4	**54**	**36**	**21**	**11**	**5**	**3**	**1**			
	59	**40**	**24**	**13**	**7**	**3**	**2**			
10**6	**62**	**44**	**27**	**15**	**8**	**4**	**2**			
	65	**47**	**30**	**17**	**9**	**5**	**2**	**1**		
10**8	**68**	**50**	**33**	**19**	**10**	**5**	**3**	**1**		
	70	**53**	**35**	**21**	**11**	**6**	**3**	**1**		
10**10	**72**	**55**	**38**	**23**	**13**	**6**	**3**	**2**		
	74	**58**	**40**	**24**	**14**	**7**	**4**	**2**		
10**12	**75**	**60**	**42**	**26**	**15**	**8**	**4**	**2**		
	76	**61**	**44**	**28**	**16**	**8**	**4**	**2**	**1**	
10**14	**78**	**63**	**46**	**29**	**17**	**9**	**5**	**2**	**1**	

Bucket Sort Efficiency $t_{startup} = 1$ $t_{data} = 0.05$

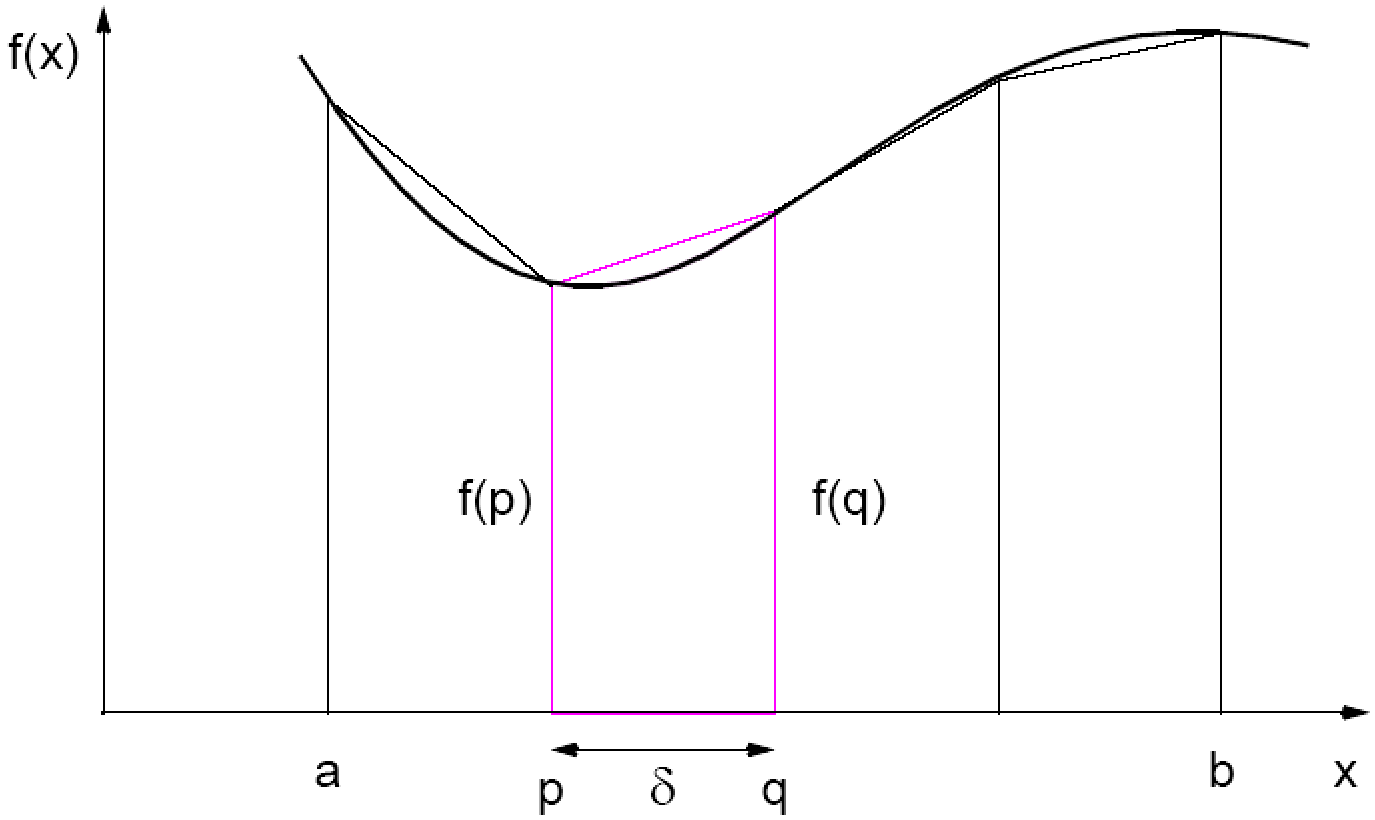
n/p	2	4	8	16	32	64	128	256	512	1024
10**4	**99**	**98**	**96**	**92**	**85**	**72**	**54**	**34**	**18**	**8**
	99	**99**	**97**	**94**	**88**	**77**	**61**	**42**	**25**	**13**
10**6	**99**	**99**	**97**	**95**	**90**	**80**	**66**	**47**	**30**	**17**
	99	**99**	**98**	**95**	**91**	**83**	**69**	**52**	**34**	**20**
10**8	**100**	**99**	**98**	**96**	**92**	**85**	**72**	**56**	**38**	**22**
	100	**99**	**98**	**96**	**93**	**86**	**75**	**59**	**41**	**25**
10**10	**100**	**99**	**98**	**97**	**93**	**87**	**77**	**62**	**44**	**27**
	100	**99**	**99**	**97**	**94**	**88**	**79**	**64**	**47**	**30**
10**12	**100**	**99**	**99**	**97**	**94**	**89**	**80**	**66**	**49**	**32**
	100	**99**	**99**	**97**	**95**	**90**	**82**	**68**	**51**	**34**
10**14	**100**	**99**	**99**	**98**	**95**	**91**	**83**	**70**	**53**	**36**

Numerical integration using rectangles

Each region calculated using an approximation given by rectangles:
Aligning the rectangles:



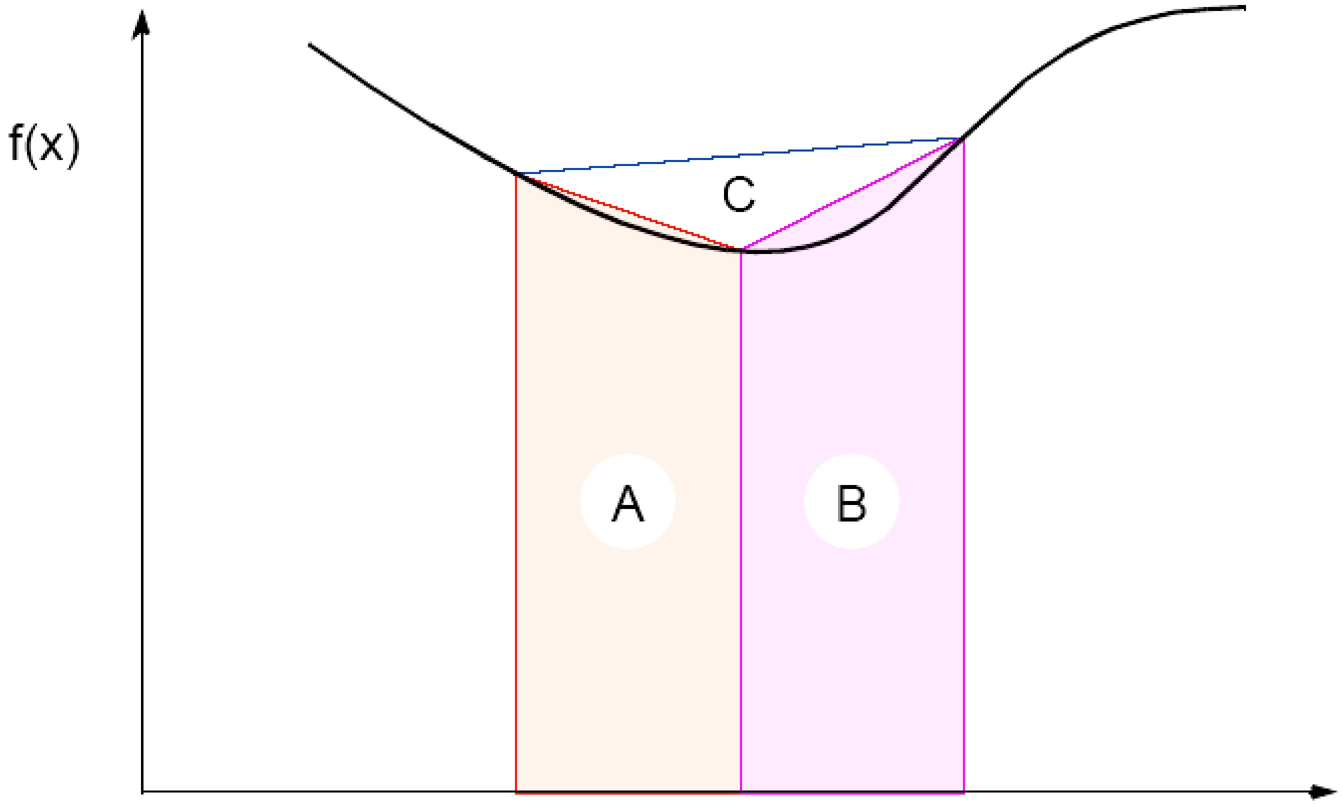
Numerical integration using trapezoidal method



May not be better!

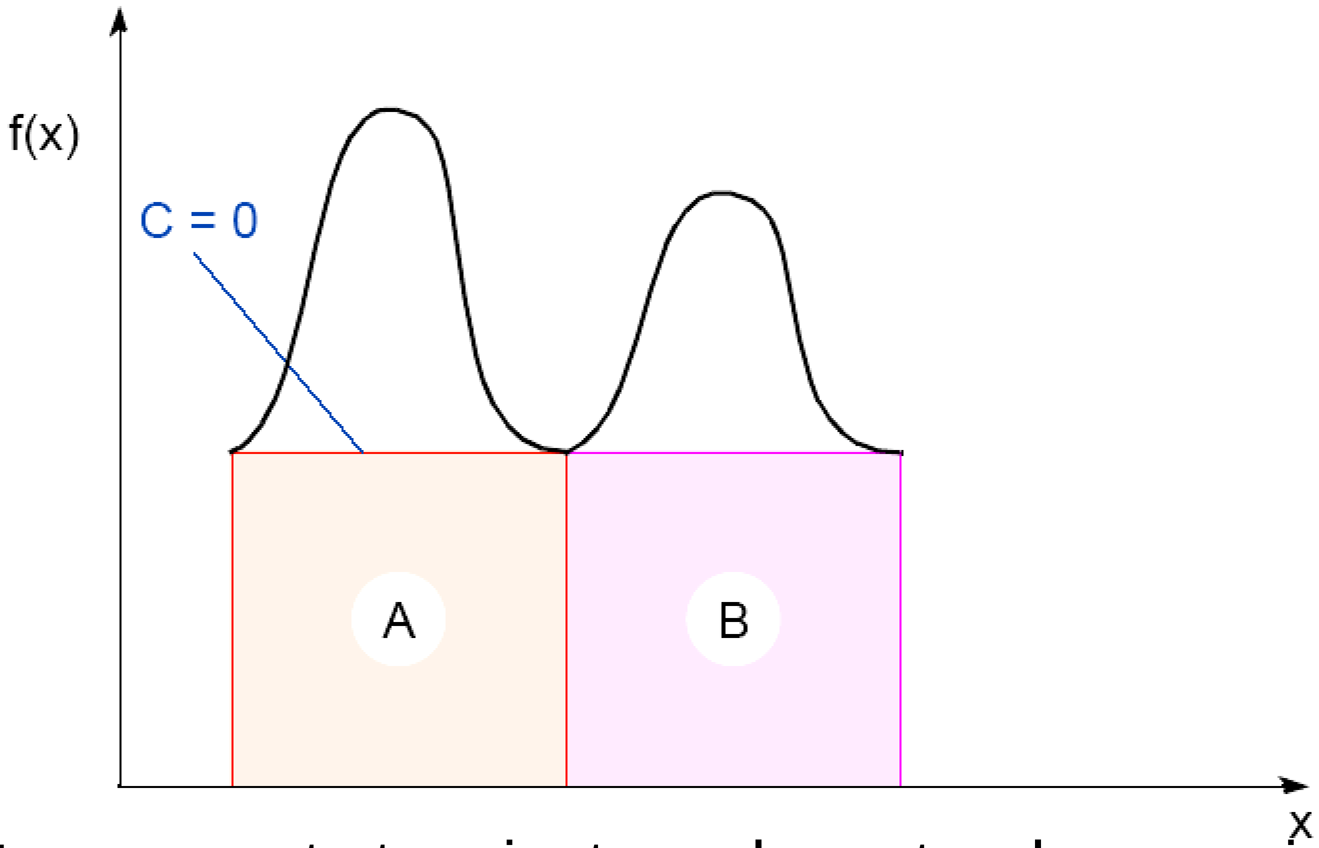
Adaptive Quadrature

Solution adapts to shape of curve. Use three areas, *A*, *B*, and *C*. Computation terminated when largest of *A* and *B* sufficiently close to sum of remain two areas .



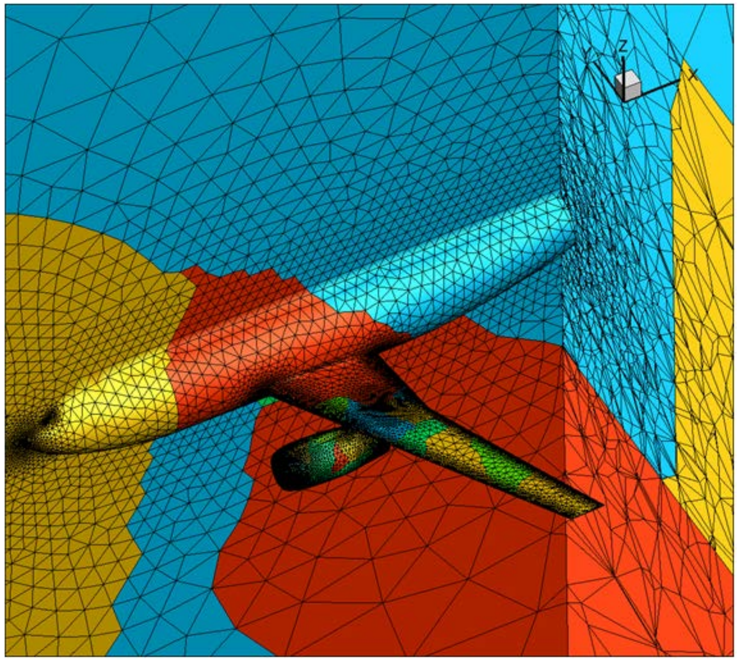
Adaptive quadrature with false termination.

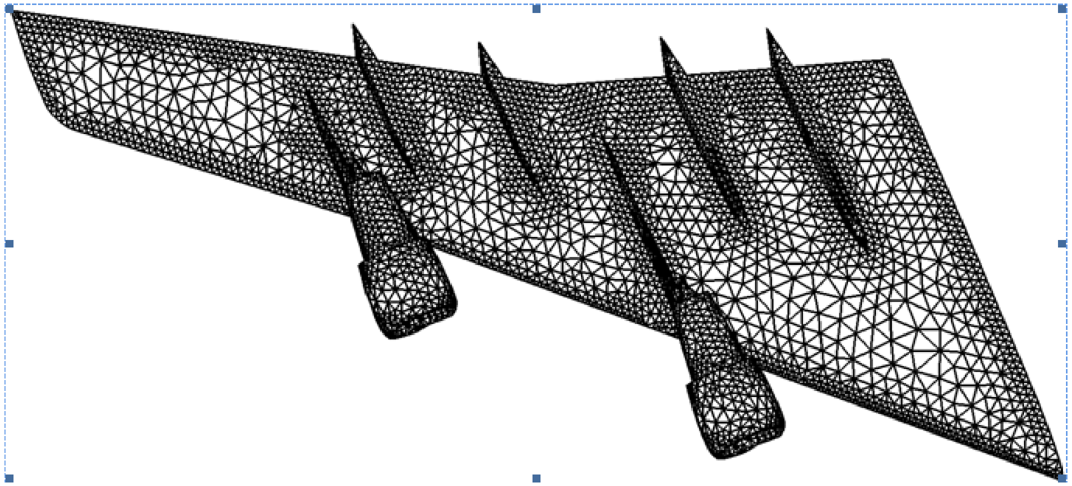
Some care might be needed in choosing when to terminate.



Might cause us to terminate early, as two large regions are the same (i.e., *C* = 0).

- Adaptive algorithms such as this are used in many scientific and engineering applications e.g. aerospace. The size of spatial elements is reduced where accuracy is needed e.g the triangles on the surface and tetrahedra in the volume around features such as on the wing below





We then need to use graph-based Techniques to partition the domain on different processors shown by different colors

Source Jianjun Chen

Simple program to compute π

Using C++ MPI routines

```
/*********************************************************************************
pi_calc.cpp calculates value of pi and compares with actual
value (to 25digits) of pi to give error. Integrates function f(x)=4/(1+x^2).
July 6, 2001 K. Spry CSCI3145
*********************************************************************************/
#include <math.h> //include files
#include <iostream.h>
#include "mpi.h“

void printit();                                         //function prototypes
int main(int argc, char *argv[])
{
double actual_pi = 3.141592653589793238462643;
                                                        //for comparison later
int n, rank, num_proc, i;
double temp_pi, calc_pi, int_size, part_sum, x;
char response = 'y', resp1 = 'y';
MPI::Init(argc, argv);                                  //initiate MPI
```

```
num_proc = MPI::COMM_WORLD.Get_size();
rank = MPI::COMM_WORLD.Get_rank();
if (rank == 0) printit();            /* I am root node, print out welcome */

while (response == 'y') {
          if (resp1 == 'y') {
          if (rank == 0) {           /*I am root node*/
          cout <<"__________________________________" <<endl;
          cout <<"\nEnter the number of intervals: (0 will exit)" << endl;
          cin >> n;}
} else n = 0;

MPI::COMM_WORLD.Bcast(&n, 1, MPI::INT, 0);    //broadcast n
if (n==0) break; //check for quit condition
```

```
else {
int_size = 1.0 / (double) n;                              //calcs interval size
part_sum = 0.0;

for (i = rank + 1; i <= n; i += num_proc)
 {                                                        //calcs partial sums
          x = int_size * ((double)i - 0.5);
          part_sum += (4.0 / (1.0 + x*x));
}
temp_pi = int_size * part_sum;
                                  //collects all partial sums computes pi

MPI::COMM_WORLD.Reduce(&temp_pi,&calc_pi, 1,
MPI::DOUBLE, MPI::SUM, 0);
if (rank == 0) {                                  /*I am server*/
cout << "pi is approximately " << calc_pi
<< ". Error is " << fabs(calc_pi - actual_pi)
<< endl
<<"_______________________________________"
<< endl;
}
}                                                 //end else
if (rank == 0) { /*I am root node*/
cout << "\nCompute with new intervals? (y/n)" << endl; cin >> resp1;
}
}//end while
MPI::Finalize();                                  //terminate MPI
return 0;
}                                                 //end main
```

```
//functions
void printit()
{
cout << "\n************************************" << endl
<< "Welcome to the pi calculator!" << endl
<< "Programmer: K. Spry" << endl
<< "You set the number of divisions \nfor estimating the
integral:
\n\tf(x)=4/(1+x^2)"
<< endl
<< "************************************" << endl;
}                    //end printit
```

Gravitational *N*-Body Problem

Finding positions and movements of bodies in space subject to gravitational forces from other bodies, using Newtonian laws of physics.

Gravitational *N*-Body Problem Equations

Gravitational force between two bodies of masses m_a and m_b is:

$$F = \frac{Gm_a m_b}{r^2}$$

G is the gravitational constant and *r* the distance between the bodies. Subject to forces, body accelerates according to Newton's 2nd law:

$$F = ma$$

m is mass of the body, *F* is force it experiences, and *a* the resultant acceleration.

Details

Let the time interval be Δt. For a body of mass *m*, the force is:

$$F = \frac{m(v^{t+1} - v^{t})}{\Delta t}$$

New velocity is:

$$v^{t+1} = v^{t} + \frac{F\Delta t}{m}$$

where v^{t+1} is the velocity at time $t + 1$ and v^{t} is the velocity at time *t*.

Over time interval Δt, position changes by

$$x^{t+1} - x^{t} = v\Delta t$$

where x^{t} is its position at time *t*.
Once bodies move to new positions, forces change.
Computation has to be repeated.

Sequential Code

Overall gravitational *N*-body computation can be described by:

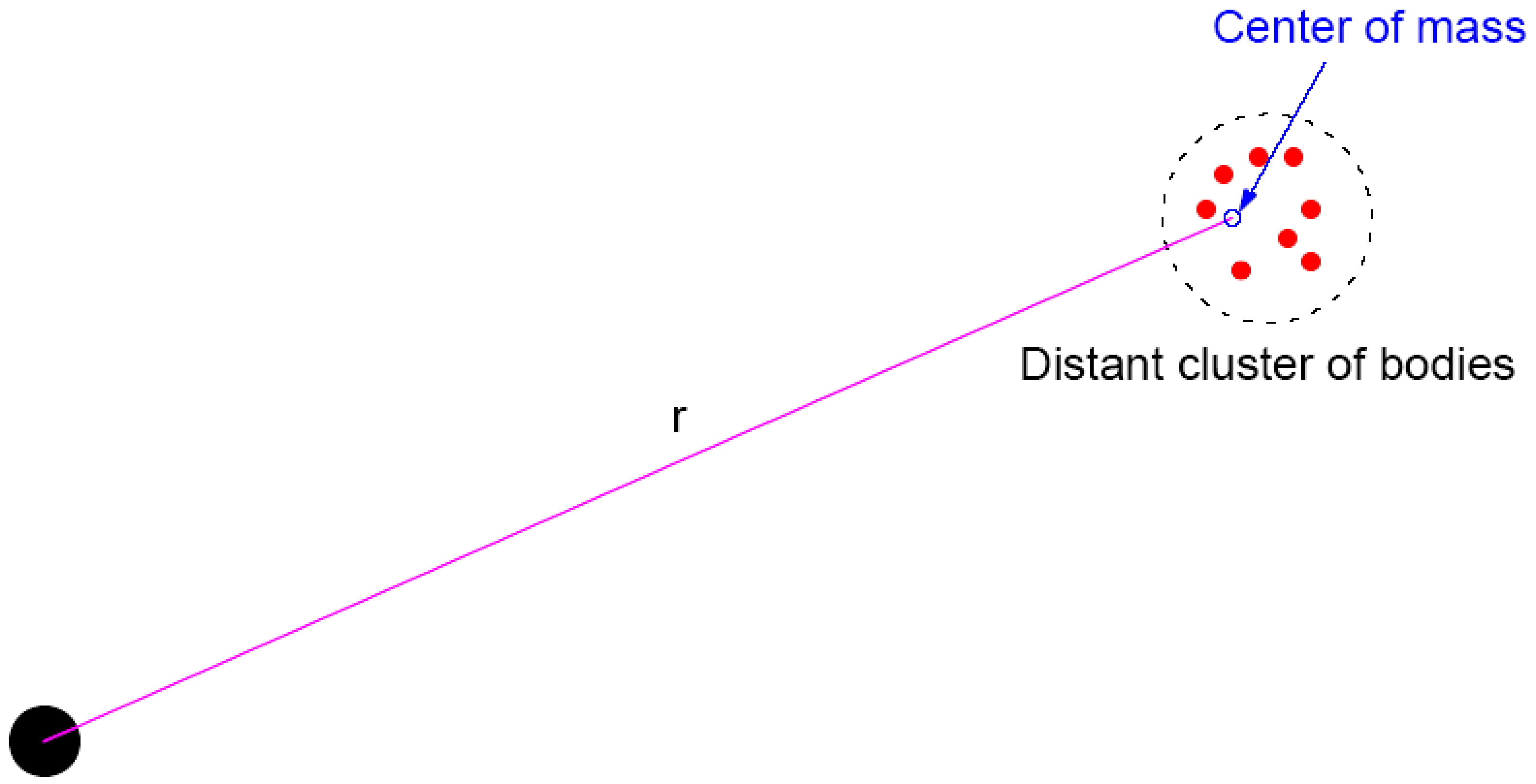
```
for (t = 0; t < tmax; t++)                  /* for each time period */
   for (i = 0; i < N; i++) {                /* for each body */
       F = Force_routine(i);                /* compute force on ith body */
       v[i]new = v[i] + F * dt / m;         /* compute new velocity */
       x[i]new = x[i] + v[i]new * dt;       /* and new position */
}
for (i = 0; i < nmax; i++) {                /* for each body */
     x[i] = x[i]new;                        /* update velocity & position*/
     v[i] = v[i]new;
}
```

Parallel Code

The sequential algorithm is an O(N^2) algorithm (for one iteration) as each of the *N* bodies is influenced by each of the other *N* - 1 bodies.

Not feasible to use this direct algorithm for most interesting *N*-body problems where *N* is very large.

Time complexity can be reduced approximating a cluster of distant bodies as a single distant body with mass sited at the center of mass of the cluster:



Barnes-Hut Algorithm

Start with whole space in which one cube contains the bodies (or particles).

- First, this cube is divided into eight subcubes.

- If a subcube contains no particles, subcube deleted from further consideration.

- If a subcube contains one body, subcube retained.

- If a subcube contains more than one body, it is recursively divided until every subcube contains one body.

Creates an *octtree* - a tree with up to eight edges from each node.

The leaves represent cells each containing one body.

After the tree has been constructed, the total mass and center of mass of the subcube is stored at each node.

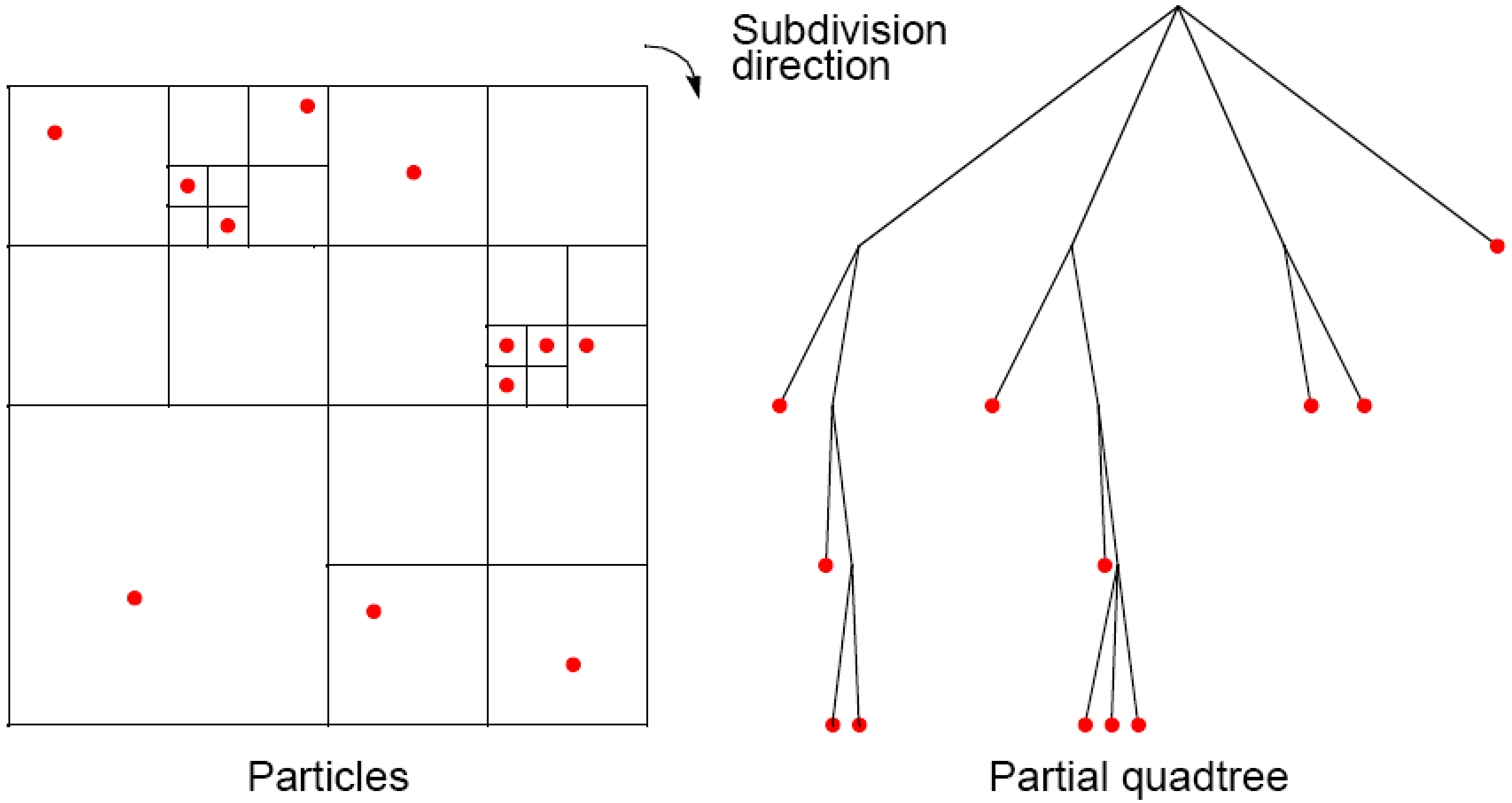
Force on each body obtained by traversing tree starting at root, stopping at a node when the clustering approximation can be used, e.g. when:

$$r \geq \frac{d}{\theta}$$

where θ is a constant typically 1.0 or less.

Constructing tree requires a time of O($n$log$n$), and so does computing all the forces, so that overall time complexity of method is O($n$log$n$).

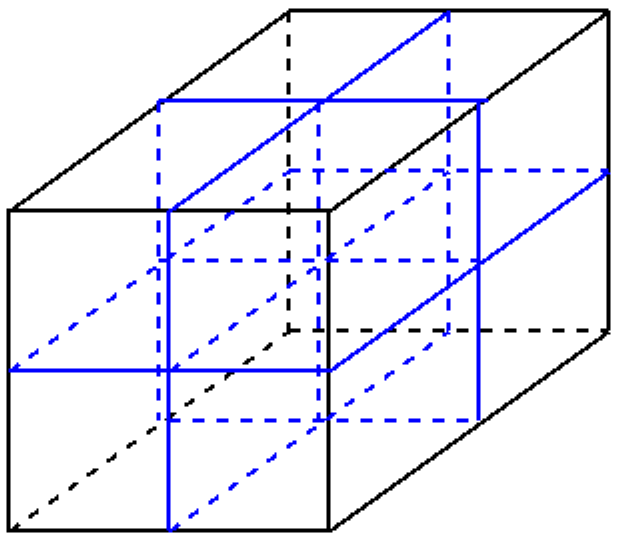
Quad tree Recursive division of 2-dimensional space

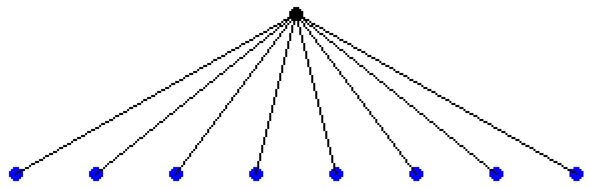


Oct Trees

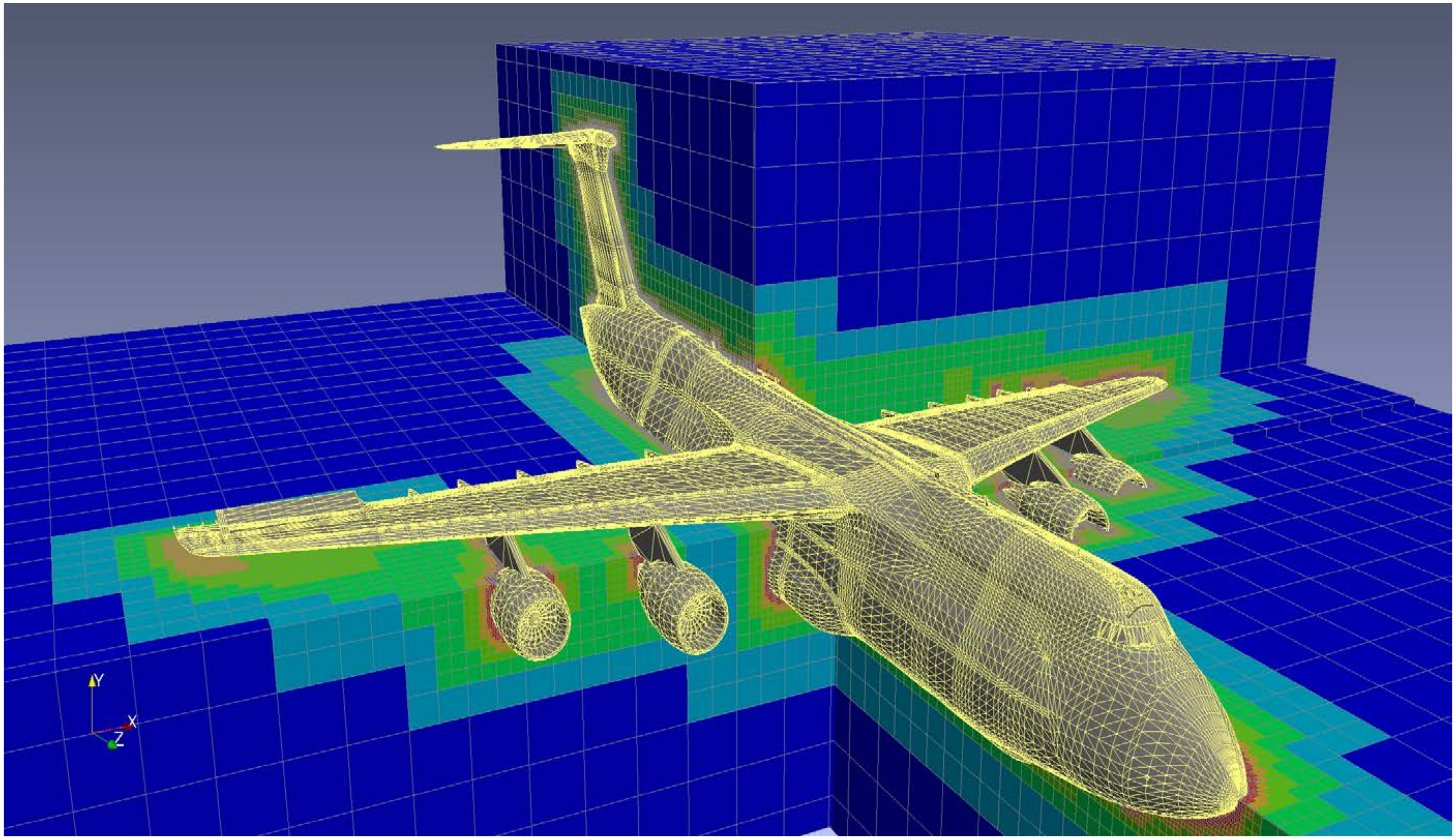
- Extension of quad tree idea to 3D.
- Very widely used in many applications in science and engineering
- Subdivide space into 8 at each step
- Adaptive subdivision can be used as in quad tree case

2 Levels of an Octree



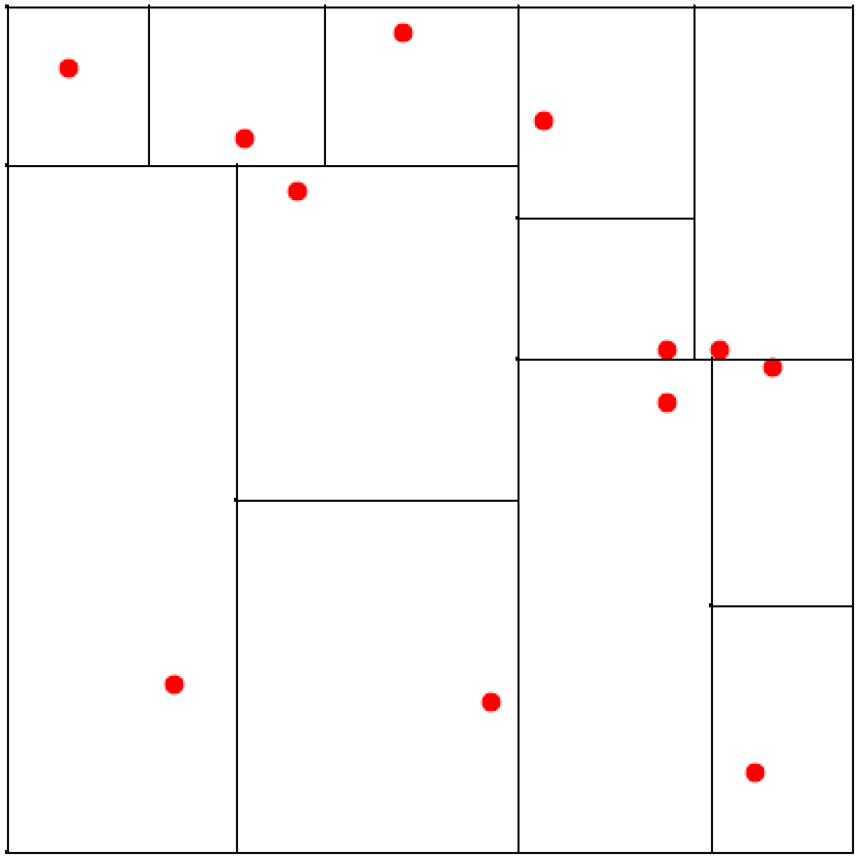


Octree Based Mesh of Aircraft



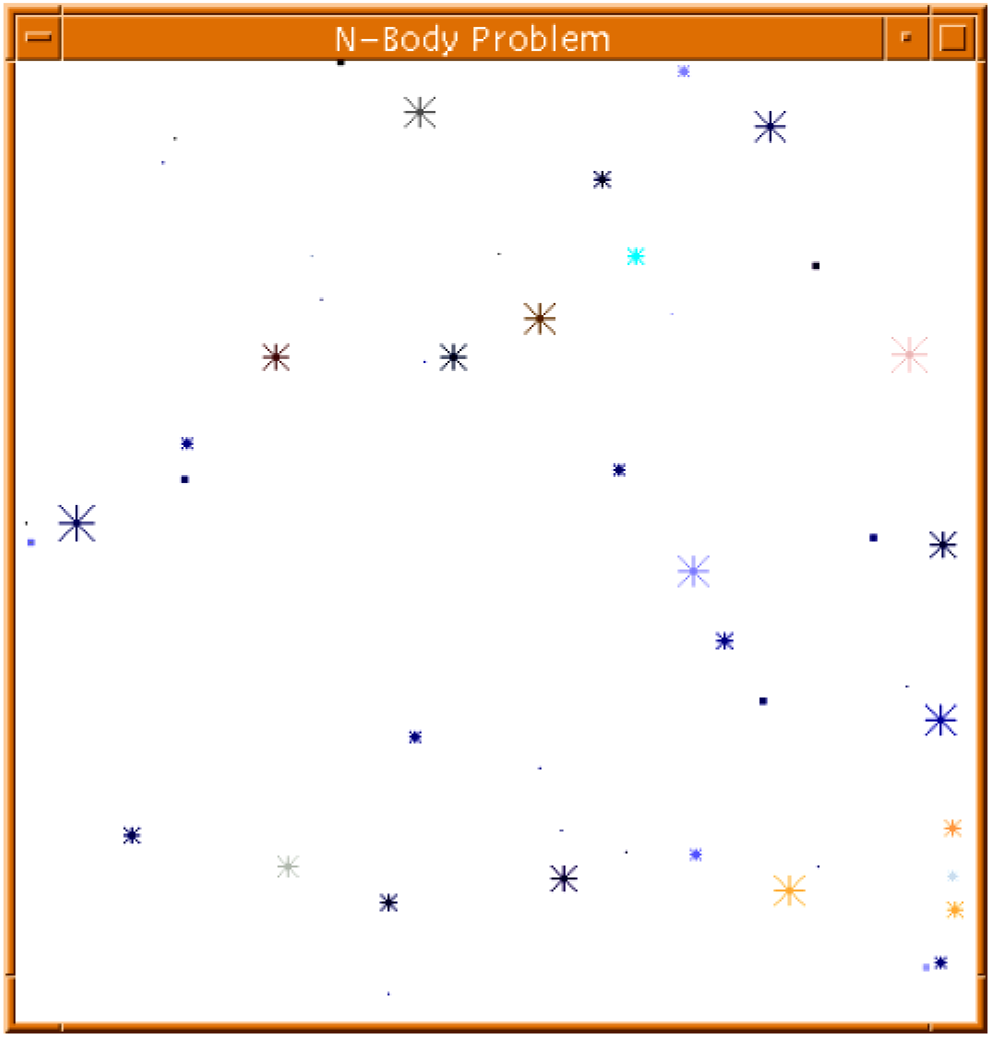
Orthogonal Recursive Bisection

(For 2-dimensional area) First, a vertical line found that divides area into two areas each with equal number of bodies. For each area, a horizontal line found that divides it into two areas each with equal number of bodies. Repeated as required.



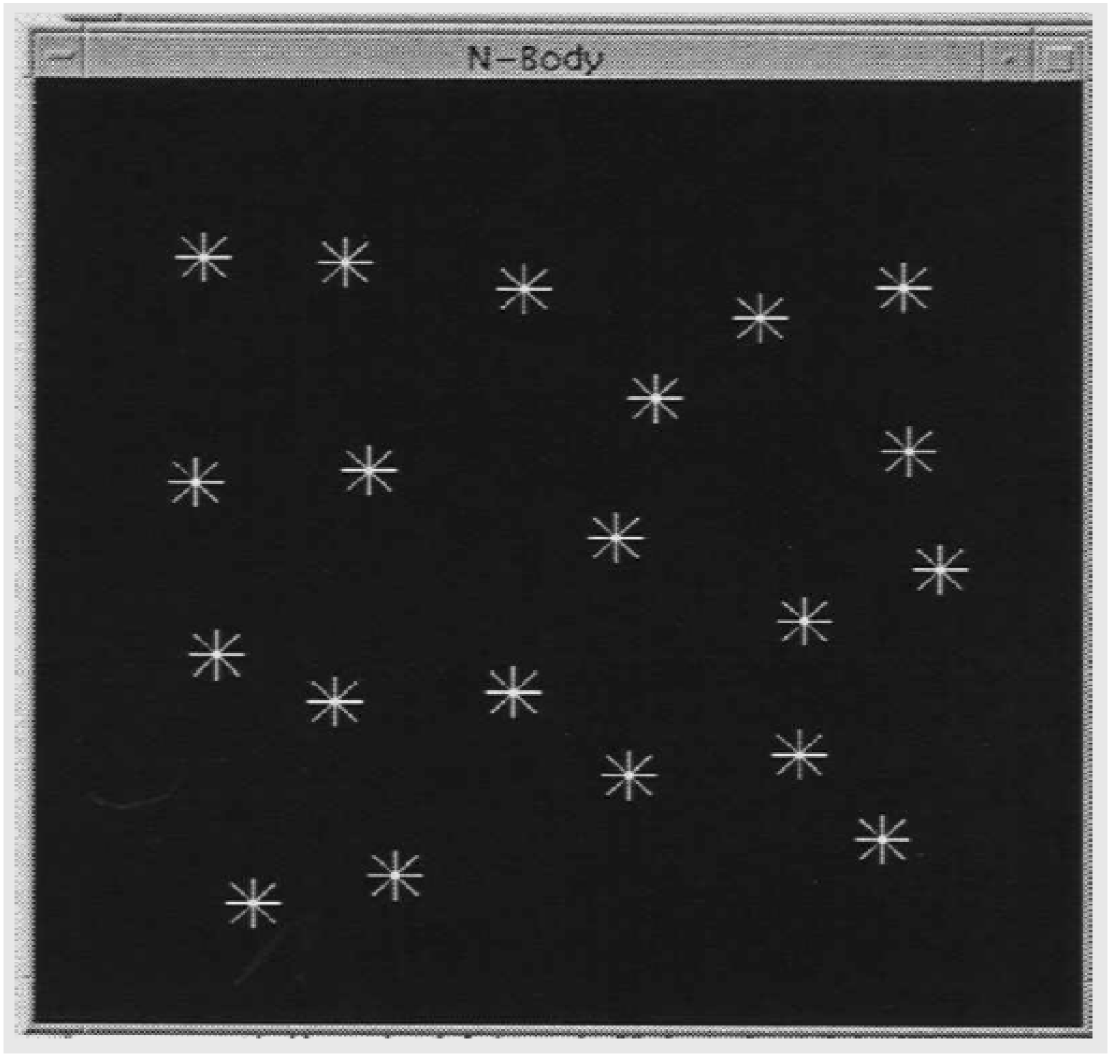
Astrophysical *N*-body simulation

By Scott Linssen (UNCC student, 1997) using O(N^2) algorithm.



Astrophysical *N*-body simulation

By David Messager (UNCC student 1998) using Barnes-Hut algorithm.



Method of Choice (?) – Fast Multipole Method FMM

- The **fast multipole method (FMM)** is a mathematical technique that was developed to speed up the calculation of long-ranged forces in the n-body problem. It does this by expanding the system Green's function using a multipole expansion, which allows one to group sources that lie close together and treat them as if they are a single source.[1]
- The FMM, introduced by Rokhlin and Greengard, has been acclaimed as one of the top ten algorithms of the 20th century.[4] The FMM algorithm dramatically reduces the complexity of matrix-vector multiplication involving a certain type of dense matrix which can arise out of many physical systems.
- the FMM has also been applied for efficiently treating the Coulomb interaction in Hartree–Fock and density functional theory calculations in quantum chemistry.
- Reduces complexity from O(n*n) to O(n) – very important for large n, constant in front n is large – good for GPUs